μPnP: Plug and Play Peripherals for the Internet of Things

Fan Yang, Nelson Matthys, Rafael Bachiller, Sam Michiels, Wouter Joosen and Danny Hughes

iMinds-DistriNet, KU Leuven, Leuven, B-3001, Belgium

{name.surname}@cs.kuleuven.be

Abstract

Internet of Things (IoT) applications require diverse sensors and actuators. However, contemporary IoT devices provide limited support for the integration of third-party peripherals. To tackle this problem, we introduce μPnP: a hardware and software solution for plug-and-play integration of embedded peripherals with IoT devices. μPnP provides support for: driver development, automatic integration of third-party peripherals, discovery and remote access to peripheral services. This is achieved through a low-cost hardware identification approach, a lightweight driver language and a multicast network architecture. Evaluation shows that μPnP has a minimal memory footprint, reduces development effort and provides true plug-and-play integration at orders of magnitude less energy than USB.

Categories and Subject Descriptors C.2.4 Distributed Systems [*Distributed applications*]

Keywords Internet of Things, Wireless Sensor Networks, Plug-and-Play, Device Drivers

1. Introduction

The Internet of Things (IoT) is moving out of the lab and into the real-world, where it is being applied at large scale in diverse application scenarios. To optimally accommodate more varied real-world scenarios, support is required for the integration of various third-party peripherals such as sensors, actuators and radios with networks of IoT devices.

The Plug-and-Play (PnP) integration of peripherals with distributed systems has received significant attention in mainstream computer systems, where the problem has been addressed through a combination of standard hardware interconnects [2, 17] and service discovery protocols [5, 14, 42]. However, these conventional approaches are inefficient in terms of energy and memory usage for resource-constrained IoT devices. For example, our evaluation platform, the ATMega128RFA1 microcontroller offers a 16MHz 8-bit core, 16KB of RAM, 128KB of flash memory and an 802.15.4 radio [6]. Resources are scarce and these devices must often operate for long periods on a tight energy budget.

In terms of *hardware support*, there are a number of standardised hardware interconnects for embedded devices, including: UART [33], SPI [30] or I2C [31]. However, these approaches lack a *device type identifier* and therefore cannot support plug-and-play device integration. In contrast, hardware interconnects for resource-rich systems such as USB [17] and Firewire [2] provide the necessary device type information to initiate plug-and-play device integration, but require that specific chips are embedded on all peripherals, which increases costs and energy consumption.

In terms of *software support*, a variety of service discovery approaches have been proposed such as: Jini [5], UPnP [42] or SLP [14]. However, these approaches do not support the development or deployment of device drivers. Instead, they assume that drivers are pre-loaded on all networked peripherals. This precludes the connection of new peripherals at runtime. Support is required for the development and remote deployment of networked device drivers.

This paper introduces μPnP which realizes PnP peripheral integration for the IoT using an integrated hardware and software approach. The *hardware element* uses passive electrical characteristics as an efficient mechanism to identify peripherals, which may use various existing peripheral interconnects, including: ADC, I2C, SPI and UART. This allows existing peripherals to be easily repackaged as μPnP devices. The *software element* of μPnP provides a lightweight platform-independent driver language, together with an efficient multicast architecture for peripheral discovery and access.

End-users of μPnP benefit from easily customized hardware peripherals and simplified management of the corresponding drivers. Developers on the other hand, benefit from the platform-independent driver language, which reduces the complexity of drivers.

The scientific contributions of this paper are three-fold. First, we contribute a *novel hardware architecture* for pluggable IoT peripherals that reduces energy consumption

EuroSys'15, April 21–24, 2015, Bordeaux, France.

http://dx.doi.org/10.1145/10.1145/2741948.2741980

in comparison to USB. Second, we provide a *platform-independent driver language* for the IoT. Finally, we contribute an efficient *multicast-based network architecture* for device discovery.

We evaluate the μPnP hardware and software approach through comparison to an embedded USB controller and standard C device drivers respectively. Our evaluation shows that μPnP is efficient in terms of: energy consumption, memory footprint and development effort. Furthermore, remote μPnP peripheral discovery performs well on embedded devices.

The remainder of this paper is structured as follows. Section 2 provides context on the problem of PnP device integration. Section 3 describes the μPnP hardware identification approach. Section 4 describes the μPnP device driver language. Section 5 describes the μPnP network architecture. Section 6 evaluates an implementation of μPnP for the Contiki [12] operating system on the ATMega128RFA1 [6]. Section 7 discusses related work. Section 8 concludes. Finally, Section 9 discusses directions for future work.

2. Problem and Motivation

When considering the problem of peripheral integration, there are many parallels between today's IoT devices and mainstream computing systems of the 1980s. Early mainstream approaches to peripheral integration such as the ISA bus [20] or UART [33] required extensive manual configuration. This included: physical jumper settings, OS settings and device driver software. This process was quickly recognised to be an onerous task, which led to the development of a range of 'Plug-and-Play' (PnP) peripheral technologies, beginning with the MIT NuBus in 1984 [1] and culminating in contemporary PnP peripheral approaches such as USB [17]. As local area networks proliferated in the 1990s, it became increasingly important to integrate peripherals as first-class networked devices. This led to the development of protocols for remote peripheral discovery and usage, such as Jini [5], UPnP [42] and SLP [14].

Contemporary approaches to integrating peripherals with the IoT face the same problems as the mainstream approaches of the early 1980s. The integration of devices remains an onerous task that involves extensive hardware and software configuration. This makes it difficult to customize IoT devices and limits the range of applications that the IoT can tackle. We envisage a different future for the IoT, where low-cost IoT devices may be easily customized with embedded PnP peripherals, empowering non-experts to build tailored IoT systems for a variety of application domains. Examples of IoT peripherals include sensors (e.g. accelerometers, microphones or RFID card readers) and actuators (e.g. relay switches, screens or speech synthesizers).

The full problem of PnP networked peripheral integration has three elements: (i.) *hardware* to identify IoT peripherals, (ii.) *software* support for the development and deployment of peripheral drivers and (iii.) *networking* support to remotely discover and use IoT peripherals. Figure 1 shows the complete PnP process and maps the contributions of μPnP against it.

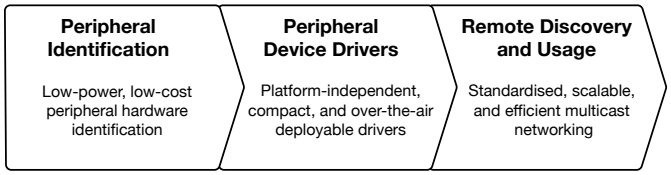

Figure 1. The PnP problem and key contributions of μPnP

2.1 Peripheral identification

Standard hardware identification approaches from resource-rich mainstream systems require custom chips such as USB [17] or Firewire [2] to be embedded on every peripheral. These approaches focus primarily upon performance, rather than; energy-consumption, computation and memory footprint. This renders them inefficient in the context of the IoT. Moreover, providing custom hardware chips for every peripheral quickly becomes costly for large networks of devices, each of which may have multiple peripherals. This is particularly problematic for the IoT community, who aim at a per-device cost of a few dollars.

μPnP contributes a novel approach to peripheral identification. First, peripherals are identified based upon their passive electrical characteristics, eliminating the need for additional hardware controllers on peripherals. Second, existing hardware interconnects (e.g. ADC, I2C, UART) are encapsulated in the μPnP bus, allowing existing peripherals to be easily repackaged as μPnP devices. μPnP hardware identifiers map each peripheral to a description in the open, global μPnP address space. The μPnP peripheral identification approach is described in Section 3.

2.2 Peripheral Device Drivers

The current state-of-practice in IoT driver development is to manually develop driver software based upon data-sheet specifications of peripheral functionality. Drivers are written in low-level programming languages such as C and are platform-specific due to the use of register manipulation, low-level routines and interrupt handlers. This leads to high development effort and code that is not reusable. Even for simple peripherals, such as an analog temperature sensor, developers must understand how to use Analog to Digital Converter (ADC) registers and be aware of ADC resolution, supply voltage and reference voltage.

μPnP tackles this problem by providing a platform-independent driver language for IoT platforms. This language relieves developers from platform-specific peculiarities and improves driver portability. μPnP drivers are compact and supported by an efficient runtime environment on every μPnP *'Thing'*. μPnP drivers provide a clean separation

of concerns between peripheral behaviour, which is encoded in the μPnP driver and platform-specific logic which is encapsulated in the supporting μPnP driver manager. In addition, μPnP allows drivers to be deployed over-the-air. The μPnP device driver architecture is described in Section 4.

2.3 Remote discovery and usage

State-of-the-art approaches to peripheral discovery [14, 42] rely on dedicated application-layer protocols to mediate service discovery and usage of networked peripherals. This results in a large software stack. Furthermore, these approaches often use verbose data representation schemes that are not well suited to resource-constrained IoT devices.

To address this problem, μPnP contributes a lightweight remote service discovery and interaction protocol that builds upon standard IPv6 multicast. By exploiting IPv6 multicast features, message passing is reduced. Furthermore, IPv6 facilities are required by default to be present in both mainstream hosts and IoT devices, which minimizes the size of the resulting software stack. The μPnP network architecture is described in Section 5.

3. μPnP Hardware Identification

μPnP uses a dedicated hardware circuit to identify peripherals. Whenever a peripheral is connected to the μPnP control board, it generates a timed pulse, which is translated into a unique identifier. This identifier maps to the global μPnP address space.

The key concept in the μPnP hardware circuit is to convert passive electrical components into a unique timed pulse. μPnP hardware uses multivibrators working in monostable mode to create pulses, whose length is determined by passive electrical components.

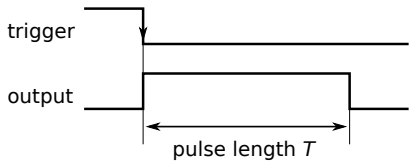

Figure 2. The multivibrator generates a pulse after being triggered by a falling edge.

Figure 2 shows an example pulse triggered by a falling edge. The pulse length T is related to a resistor R and a capacitor C. k is a constant. The passive elements R or C determine the length of the timed pulse as follows:

$$T = k \times R \times C \tag{1}$$

Each μPnP peripheral embeds a unique set of resistors. While the μPnP control board contains a set of multivibrators and fixed value capacitors. When a peripheral is connected to the control board it thus generates a pulse of a specific length that is used to identify

Neither resistors nor capacitors are precisely calibrated and when used to represent discrete categories the required component values grow exponentially due their inherent inaccuracy. [21]. To avoid the pulse length becoming too long, μPnP uses a series of 4 short pulses instead of one long pulse to identify each sensor. This approach keeps the worst-case pulse length short, while accounting for the inherent inaccuracy of passive components.

To generate the required pulses, 4 multivibrators are connected in serial, such that each multivibrator generates a pulse, which is also used as a trigger for the next multivibrator. As can be seen in Figure 3, a unique sensor ID is defined by 4 time intervals (T1-T4), each of which is mapped to a single byte value, thus resulting in a 32-bit address for each device type and allowing for over 4 million unique device type identifiers.

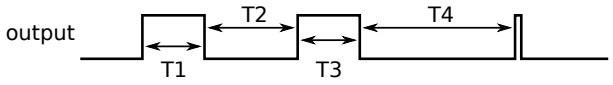

Figure 3. The waveform for one sensor is constructed of 4 time intervals: T1, T2, T3 and T4.

3.1 μPnP Peripheral Hardware

As described above, each peripheral identifier is defined by the four pulse time intervals (T1 to T4 in Figure 3). In practice, resistors are more precise and cost much less than capacitors, so a set of capacitors of fixed value are used on the control board and only the resistor values embedded on each μPnP peripheral must be customized.

The current prototype of μPnP uses a 19-pin mini HDMI connector to connect peripherals. Pin 1 to pin 8 are used for the μPnP *identification* scheme, and pin 10 to pin 12 are assigned to *communication* with the embedded device. The μPnP control board supports *multiplexing* of hardware interconnects; based upon the detected device ID, as shown in Table 1, the communication pins (pin 10 to pin 12) are switched to the appropriate communication bus. At the present time, μPnP supports common microcontroller interconnects, such as: ADC, I2C, SPI and UART. Figure 4 shows an abstracted version of the device identification circuit, the μPnP hardware interconnect and its mapping to each communication bus. Circuit diagrams and associated files are available online at: `www.micropnp.com`. This includes schematics for an Arduino-compatible control board and a peripheral prototyping board.

Figure 4 shows the peripheral identification circuit, which is composed of just 4 resistors. This ensures that anyone with a basic knowledge of electronics can begin building their own μPnP peripherals. Table 1 shows how the μPnP connector maps to different peripheral interconnects.

3.2 μPnP Control Board Hardware

The control board works as an interconnect between the MCU and μPnP peripherals. Based upon the concepts intro-

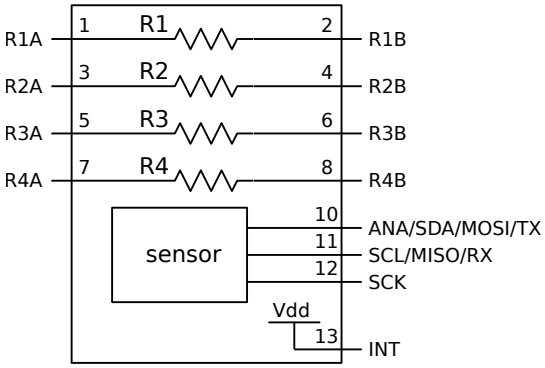

Figure 4. μPnP uses only 4 resistors on the sensor side to encode device IDs, minimizing complexity.

Bus	Pin10	Pin11	Pin12
ADC	Analog Signal	N/C	N/C
I2C	SDA	SCL	N/C
SPI	MOSI	MISO	SCK
UART	TX	RX	N/C

Table 1. Pinout for different communication bus interfaces. (N/C = Not Connected)

duced above, the μPnP hardware is constructed using multivibrators. Each peripheral requires 4 multivibrators to generate a device ID for each channel. To minimize board-size and cost, the μPnP control board uses the same set of multivibrators to generate IDs for each channel and separates these channels in a time sequence.

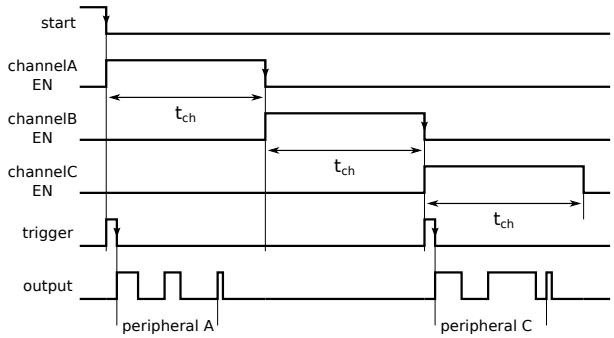

Figure 5. Each μPnP channel is enabled for a discrete timeslot, allowing them to share the same set of multivibrators.

As can be seen from Figure 5, each channel is enabled for a discrete time period t_{ch}, after which the next channel will take over the multivibrators. Figure 5 shows the waveform when peripherals are connected to channel A and C, while channel B remains unconnected. This design minimizes the number of components required and ensures that all sensor IDs are dasy-chained on one signal. This means that only three IO pins are required to interface with the μPnP control board. One pin is used to trigger the process (start in Figure 5), the second one reads the device IDs (output in Figure 5), and the last one provides an interrupt when a device is connected or disconnected. Following identification of the embedded peripheral, its communication pins are switched to the appropriate communication bus (ADC, UART, SPI or I2C).

The μPnP control board consumes a non-trivial amount of power (in the case of our prototype, an average of 7 mA at 3.3V). We minimize power consumption by adding an interrupt circuit to the control board. Receipt of an interrupt causes power to be supplied to the μPnP control board and prompts the master microcontroller to run the peripheral identification software routine. The control board is therefore only activated from the time when a new peripheral is connected or disconnected (causing an interrupt) until all connected peripherals have been identified. The average power draw of μPnP is therefore low and scales linearly with the frequency at which peripherals are connected and disconnected. Figure 6 provides a logical block diagram for the μPnP control board. Figure 7 shows a μPnP control board implemented as an Arduino shield, with two peripheral boards connected.

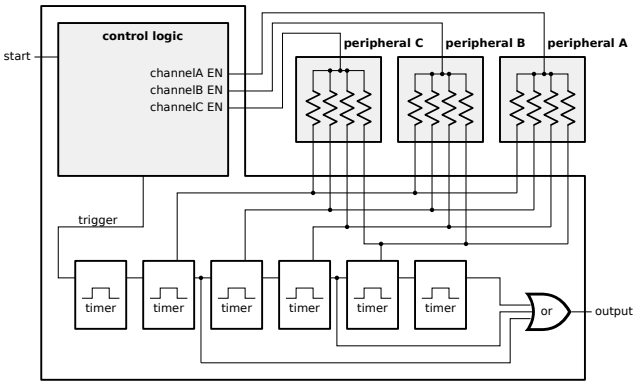

Figure 6. Logical block diagram of the μPnP control board

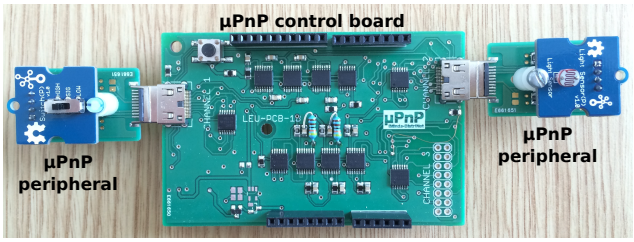

Figure 7. μPnP control board implemented as a standard-size Arduino shield and two μPnP peripherals.

The μPnP hardware approach is open source. Implementation instructions, circuit diagrams and schematics for an Arduino [8] implementation of μPnP are available online.

3.3 μPnP Global Address Space

All μPnP peripheral identifiers map to an open global address space, which is maintained at `www.micropnp.com`. The

organisation and maintenance of this address space is simple. Any party may request a *provisional address* by providing their: name, organization, email address and a link to a web resource describing the peripheral type. A simple online tool then generates the resistor set that is required to encode the assigned device identifier on the μPnP peripheral by customizing the four resistors shown in Figure 4.

A peripheral address remains provisional until a μPnP device driver is uploaded for the specified peripheral and validated, at which point it becomes a *permanent address*. At this point, the address allocation becomes immutable. However, the device drivers associated with an address may be updated at any time. Provided device drivers are integrated into the μPnP repository, allowing for remote deployment on compatible devices.

At the current time, manual checking is used to ensure the validity of peripherals and drivers. The allocation of peripheral addresses and the garbage collection of old addresses remains an area for future work.

4. μPnP Device Drivers

μPnP offers a Domain-Specific Language (DSL) that enables the implementation of driver functionality in a high-level and platform-independent manner. This language is supported by an execution environment that provides native hardware interconnection libraries for access to interconnects such as: ADC, UART and I2C. Drivers written in the μPnP DSL may be deployed Over The Air (OTA) to any μPnP Thing.

In contrast to current approaches, μPnP provides a clean separation of concerns between peripheral logic and platform logic. The μPnP DSL captures peripheral-related information that is currently provided as human-readable data sheets from peripheral manufacturers. The μPnP execution environment provides support for platform-specific concerns. This separation allows peripheral developers to focus on implementing their peripheral driver, while being shielded from platform-specific concerns. Conversely, platform developers have only to port the μPnP execution environment to make their device compatible with all μPnP peripherals.

4.1 μPnP Device Driver Language

The μPnP Domain-Specific Language (DSL) is *typed* and *event-based*. Its syntax is inspired by the simplicity and generality of the Python programming language.

Event-based programming is a natural fit with the inherently asynchronous and interrupt-driven nature of I/O operations from IoT drivers. Moreover, the event-based interaction style is commonly found within many embedded IoT operating systems [12, 16, 41] as it decouples system elements and maximises concurrency while maintaining a low memory footprint. As such, all I/O operations in μPnP are modelled as events. Device drivers may both produce and consume events, which can be generated by the driver software itself, or by the μPnP runtime, which publishes events from connected peripherals.

```
1  import uart;
2
3  uint8_t idx, rfid[12];
4  bool busy;
5
6  event init():
7    # 9600 baud, no parity, 1 stop bit, 8 data bits
8    signal uart.init(9600, USART_PARITY_NONE,
9                USART_STOP_BITS_1, USART_DATA_BITS_8);
10   idx = 0;
11   busy = false;
12
13 event destroy():
14   # restore uart to platform defaults
15   signal uart.reset();
16
17 event read(): # operation exposed over network
18   if !busy:
19     busy = true;
20     signal uart.read(); # initiate read operation
21
22 event newdata(char c):
23   # ignore CR, LF, STX, and ETX characters
24   if !(c==0x0d or c==0x0a or c==0x02 or c==0x03):
25      rfid[idx++] = c;  # store character
26   # complete RFID card ID read over uart
27   if idx == 12:
28     signal this.readDone();
29
30 event readDone():
31   busy = false;
32   idx = 0;
33   return rfid;
34
35 error invalidConfiguration():
36     signal this.destroy();
37
38 error uartInUse():
39     signal this.destroy();
40
41 error timeOut():
42    busy = false;
43    idx = 0;
```

Listing 1. Example of a UART-based device driver for the ID-20LA RFID card reader [19]

Listing 1 illustrates how a driver for a UART-based RFID card reader is implemented using our DSL. As shown, every driver defines a series of event handlers that can be invoked asynchronously. An event handler is defined using the **event** keyword, followed by a corresponding event type which triggers its execution. These event handlers run to completion and are executed atomically. The μPnP DSL does not allow

for the invocation of blocking statements as this would compromise concurrency. Lengthy operations that involve active waiting for an I/O result should thus be split into multiple parts with an explicit request, followed by a dedicated event handler to process the response. This split-phase mode of operation allows for high degrees of concurrency and performance while retaining a low memory footprint.

Control flow: All μPnP drivers must implement at least two event handlers: **`init`** and **`destroy`**. An init event (line 6) is automatically fired by the μPnP runtime when a new peripheral is plugged in and its corresponding driver is installed. Similarly, a destroy event (line 13) is sent to the driver when the peripheral is unplugged. The **`signal`** keyword, taking as arguments a destination plus event type, is used to transfer control (i.e. sending a message) between different event handlers in the driver itself as denoted using the **`this`** keyword (line 28), or to exchange an event between the driver and the supporting execution environment (line 8).

Peripheral communication: Drivers import necessary hardware interconnect functionality exposed by native libraries in the μPnP runtime using the **`import`** keyword (line 1). These libraries export a number of event handlers that can be invoked from drivers. In addition, drivers may define an arbitrary number of **`static variables`** to store state that can be manipulated in event handlers (line 3–4).

Error Handling: I/O errors, such as timeouts or invalid interconnect configurations (lines 35–43), are also modelled as events and therefore require dedicated handlers. Regular events in μPnP are handled on a first-come, first-served (FIFO) basis, while error events are prioritized. The **`error`** keyword is used to define event handlers for error messages.

Remote Usage: Drivers may expose three operations to μPnP clients over the network: **`read`**, **`write`**, and **`stream`**. Hence, depending on the underlying peripheral type, remote clients may call read (line 17; marked in red), write, or stream on a particular driver. The **`return`** keyword is used to copy and asynchronously transfer results back to clients (line 33). Section 5.3.1 discusses these operations in more detail.

Compilation: Finally, to ensure platform independence, peripheral drivers are compiled into platform-independent bytecode instructions which are interpreted by the μPnP runtime environment. In addition, in comparison to drivers that are compiled into platform-dependent native code, this typically results in very compact drivers that are more efficient to distribute and install on constrained IoT devices. While using a bytecode-based encapsulation scheme introduces an extra layer of indirection that has implications in terms of execution performance, previous research has introduced several optimization mechanisms that make this performance trade-off acceptable for embedded IoT devices [10, 24, 25].

The design of μPnP's bytecode instruction set was inspired by the Java Virtual Machine, however, it is less extensive and more tailored towards the domain of IoT driver development. The μPnP DSL compiler transforms high-level device drivers into compact bytecode instructions, allowing for energy-efficient distribution in networks of IoT nodes. Every bytecode instruction in μPnP is 8-bits in length, followed by zero or more operands.

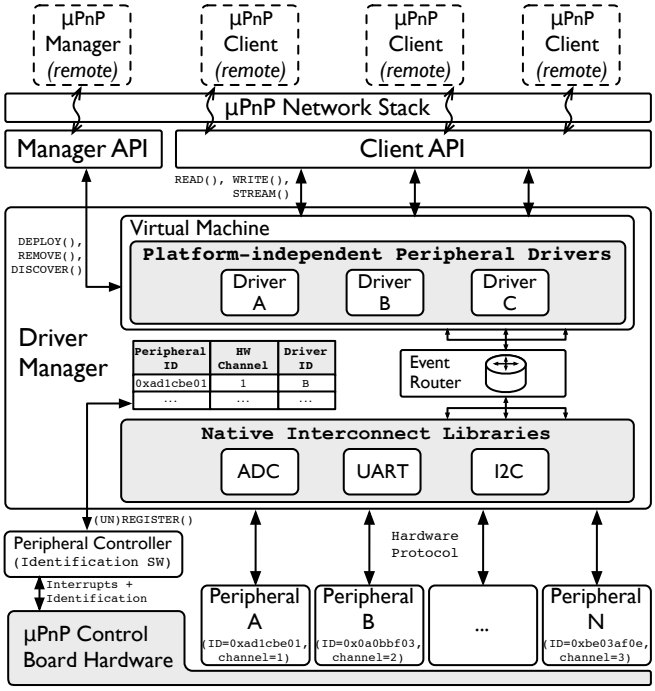

Figure 8. Overview of the μPnP Execution Environment

4.2 μPnP Execution Environment

Figure 8 illustrates μPnP's runtime environment. This consists of five major software elements: the peripheral controller, driver manager, a virtual machine, native interconnect libraries and an event router.

The **peripheral controller** interfaces with the μPnP control board and implements the hardware identification algorithm. Peripheral connection or disconnection is detected based upon an interrupt. The peripheral identification circuit is then activated and the timed pulse that results is read via a digital I/O pin. These timings are then converted to a set of 32-bit identifiers, one per channel as described in Section 3.

The **driver manager** interfaces with the peripheral controller and keeps track of the peripherals and drivers that are available. This module also integrates closely with the μPnP network stack and provides operations that enable remote deployment and removal of device drivers.

A **virtual machine** implementing a stack-based execution model executes driver bytecode. This virtual machine implements a single operand stack and concurrency is realized through event-based programming as described in Section 4.1. Stack-based architectures are known to provide a simple and memory-efficient approach for implementing virtual machines in resource-constrained systems [25, 37]. In

addition, as all concurrency is event-based, complex locking and context switching mechanisms are not required.

A set of **native interconnect libraries** implement all low-level platform specific I/O calls that are required to access local peripheral interconnects such as ADC, UART, or I2C. Every library exposes its API towards drivers as a series of standard event handlers that can be invoked by drivers as described in Section 4.1 .

Finally, the **event router** module handles the event exchange between drivers, native interconnect libraries and the network stack. As discussed previously, event handlers are executed asynchronously and do not block. To enable this, the router implements two queues: a regular FIFO queue for event processing and a priority queue for dispatching error messages. When an event is placed inside a queue, control is immediately transferred back to the originator.

5. μPnP Network Architecture

The μPnP network architecture allows for remote discovery and usage of peripherals. The architecture is composed of three software entities. The *μPnP Thing* software runs on embedded IoT devices with locally connected μPnP hardware. This software allows connected peripherals to be remotely discovered and used by μPnP-enabled clients. The *μPnP Client* software may run on both embedded IoT devices and standard computing platforms. It allows for remote discovery and interaction with μPnP Things. The *μPnP Manager* runs on a server-class device and manages the deployment and remote configuration of device drivers on μPnP Things.

All software entities are interconnected at the network layer by IPv6, which allows for the use of various low-level network technologies (e.g. Ethernet, WiFi or IEEE 802.15.4). μPnP Things are initially assigned unicast IPv6 addresses, while the μPnP manager is assigned an anycast IPv6 address to allow for network-level redundancy and scalability as described in [3]. μPnP then creates and maintains an IPv6 multicast group for each device type present in the network[1].

5.1 Multicast Addressing Schema

Each μPnP Thing generates an address for each connected peripheral and joins the multicast group(s) associated with its connected peripherals. This allows for efficient filtering of discovery traffic by peripheral type. Multicast addresses are unicast-prefix-based as defined in [15], which allows the schema to be used in either a global or local context. The multicast address schema is shown in Figure 9. The first 32 bits of all the multicast addresses are set to the standard prefix `0xff3e0030`. The following 48-bits contain the network prefix. The next 16-bits are padded with zeros (e.g `0x20010db80000000`). The last 32 bits contain the peripheral type identifier, which is generated by the μPnP hardware as described in Section 3.1. Two types of address have been reserved for special purposes: (a) the value `0x00000000` is reserved to represent all peripherals and (b) the value `0xffffffff` is reserved to represent all μPnP clients.

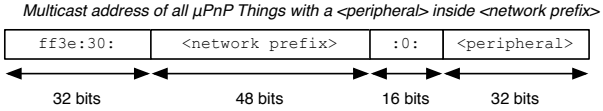

Figure 9. μPnP multicast address schema

μPnP is independent of multicast group membership and routing protocols. Our prototype implementation uses SMRF [32] as described in Section 6.

5.2 μPnP Interaction Protocol

This section describes the μPnP protocol, with a focus on the messages that mediate peripheral discovery and usage. All messages are sent as UDP packets to port 6030. The source and destination addresses of the messages not only support routing, but also determine the meaning of each message. All messages carry a unique 16-bit unsigned sequence number which is used to associate request and reply messages.

5.2.1 Peripheral Advertisement and Discovery

The peripheral discovery process is shown in Figure 10.

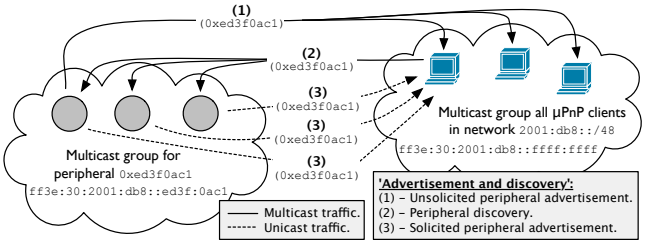

Figure 10. Peripheral advertisement and discovery

(1) Unsolicited peripheral advertisements are generated by a μPnP Thing whenever a new peripheral is connected or disconnected. The source address of this message type is the unicast IPv6 address of the μPnP Thing and the destination address is the multicast address of all μPnP clients as it is shown in Figure 10. The message is composed of a type followed by a a repeating set of fields for each locally connected peripheral. These fields are: (a) the type of sensor (fixed length of 4 bytes) and (b) a set of type-length-value (TLV) encoded tuples containing extra information about each peripheral.

(2) Peripheral discovery messages allow μPnP clients to remotely find a peripheral. The source address of these messages is the unicast IPv6 address of the μPnP client and the destination address is the multicast address of all μPnP Things with the required type of peripheral (Figure 10). The message is composed of the type followed by an

[1] For brevity, all IPv6 addresses listed in this paper are shortened using the standard representation rules specified in [22].

optional list of TLV-encoded tuples, that may be used to encode additional information about the required peripheral.

(3) Solicited peripheral advertisements are sent in response to discovery messages. These messages are syntactically equivalent to unsolicited advertisements, but the destination address is set to the unicast IPv6 address of the associated discovery message, rather than the multicast address of all μPnP clients.

5.3 Driver Management

Once a peripheral is plugged into the μPnP Thing, the platform-independent peripheral driver is installed from a μPnP manager if it is not already locally available. The driver management process is shown in Figure 11.

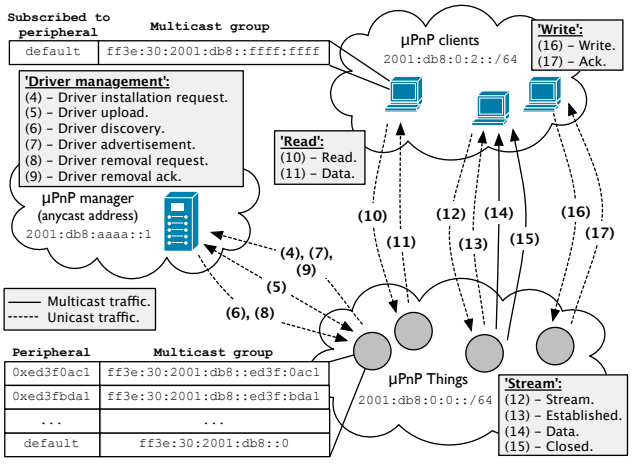

Figure 11. The μPnP management and peripheral interactions

The driver installation process sends a **(4) driver installation request** message to the anycast address of the μPnP manager. Where the μPnP manager has a driver for the specified peripheral, it initiates the **(5) driver upload** process establishes a connection with the μPnP Thing and directly uploads the driver. Then, the μPnP runtime activates the driver as described in Section 4.1. Finally, the μPnP Thing joins the multicast group indicated by the connected peripheral identifier and starts listening for μPnP messages.

The driver discovery process allows the μPnP managers to explore the set of drivers installed on a μPnP Thing. For this purpose, any μPnP manager sends a **(6) driver discovery** message to μPnP Thing and waits for a **(7) driver advertisement** response from the μPnP Thing. A μPnP manager may also remove a driver from a μPnP Thing. In this case, the μPnP manager send a **(8) driver removal** message to the μPnP Thing and waits for a **(9) driver removal acknowledgement** from the μPnP Thing.

5.3.1 Reading and Writing Data

μPnP defines two operations for data *production* that are available to μPnP clients: (a) reading a single value and (b) reading a stream of values. μPnP also defines one operation for writing a value to a peripheral (e.g. an actuator) from a μPnP client. All three operations (Figure 11) are inspired by standard file operations.

(10) Read requests allow a μPnP Thing to read a single value from a peripheral. This message is sent to the unicast IPv6 address of the μPnP Thing and the payload therefore contains the target peripheral identifier. A μPnP Thing that receives this message, responds with a **(11) data** message containing the result.

(12) Stream requests are used to subscribe to a continuous data stream produced by a peripheral. As with a *read*, this message is sent to the unicast IPv6 address of the μPnP Thing and contains the target peripheral identifier. The μPnP Thing first replies with an **(13) established** message containing the address of the multicast group that the μPnP client should join to receive the value stream. Once the μPnP client is subscribed to the group, it will begin receiving **(14) data** containing values from the peripheral. In the case that the μPnP Thing stops streaming, it sends a **(15) closed** message to the multicast group to notify all μPnP clients.

(16) Write requests from a μPnP client allow a μPnP Thing to control individual peripheral devices. As before, this message is sent to the unicast IPv6 address of the μPnP Thing and contains the target peripheral identifier. A **(17) acknowledgement** message is sent back to the μPnP client to confirm the establishment of the new value.

6. Implementation and Evaluation

We developed a prototype of the μPnP control board as an Arduino *shield* (i.e. expansion board) and four prototype peripherals: the TMP36 ADC temperature sensor [4], the HIH-4030 ADC humidity sensor [18], the ID-20LA UART RFID card reader [19] and the BMP180 I2C barometric pressure sensor [9], all from the Grove embedded peripheral range [35]. The prototype shield and two example peripherals are shown in Figure 7. At the time of writing, the overall cost of the control board is approximately 6$ (US). As only 4 resistors are needed on the peripheral side, the extra cost for each peripheral is less than 1¢ (US).

Our IoT evaluation platform is the Logos Electromechanical Zigduino [26], an Arduino-compatible microcontroller based around the ATMega128RFA1 microcontroller, which offers a 16MHz 8-bit core, 16KB of RAM, 128KB of flash memory and an 802.15.4 radio [6]. Our software implementation of the μPnP execution environment builds on the Contiki operating system version 2.7 [12].

In terms of our network stack, IPv6 support is realized through 6LowPAN [36] and the IPv6 Routing Protocol for Low-Power and Lossy Networks (RPL) [43]. Multicast routing and group management for IoT devices are provided by Stateless Multicast RPL Forwarding (SMRF) [32] and RPL, respectively.

Sections 6.1 to Section 6.4 evaluate the μPnP hardware, software stack, driver development efforts, and networking approaches respectively.

6.1 Hardware Energy Analysis

In this section, we isolate the energy consumption of the μPnP hardware. To achieve this, the μPnP control board is powered independently from a host microcontroller and the power consumption of the μPnP hardware is measured using Ohm's law based upon the voltage drop over a shunt resistor placed in series with the power supply of the control board. In our experiments we use a high precision 10 Ω resistor with a maximum relative error of 0.1%. In the case of the USB controller energy consumption is based upon the minimum idle power consumption of the USB host controller [28] and therefore represents the worst-case energy comparison for μPnP.

As explained in Section 3, the length of the identifying signal varies depending on the resistors used on peripheral boards, which leads to different energy consumption. For each identification process, the time required varies between 220 ms and 300 ms. The energy consumption therefore has a minimum value of $2.48 \times 10^{-3} J$ and a maximum value of $6.756 \times 10^{-3} J$.

To illustrate the energy impact of μPnP in a realistic setting, we simulate a one-year IoT deployment. In this deployment, we compare the energy consumption of an Arduino USB host shield [28], against the energy consumption of the μPnP shield when connected to ADC, I2C, and UART-based peripherals. Peripherals communicate once every ten seconds. We model an *ideal* peripheral which consumes no energy except for communication as this is the worst-case scenario for μPnP in terms of energy overhead.

Figure 12 presents the results of this simulation. The vertical axis shows the energy consumption over a period of one year and is measured in Joules. The horizontal axis shows the rate of change of peripherals (i.e. how often they are connected and disconnected) and is measured in minutes. It should be noted that both axes use a logarithmic scale.

As can be seen from Figure 12, μPnP consumes significantly less energy than USB in all cases and energy consumption scales linearly with the rate of change of peripherals. For example, in a situation where peripherals are changed on an hourly basis, the energy consumption of μPnP is over four orders of magnitude lower than the USB shield.

The variance captured by the error bars shown in Figure 12 arises primarily as a result of the resistor values selection on the peripheral board, as described in Section 3. Power results for the different embedded interconnects tend to diverge at low rates of peripheral change as the energy consumption due to μPnP itself becomes less significant than the power consumed by the embedded interconnect.

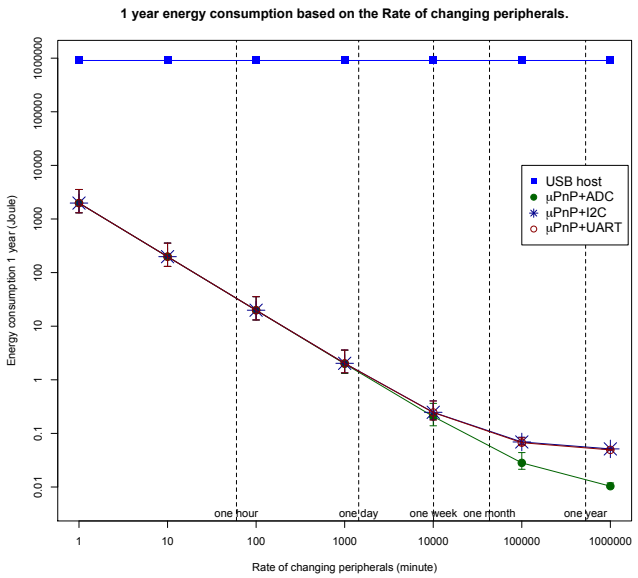

Figure 12. Energy consumption of USB versus μPnP combined with ADC, I2C, and UART interconnects

6.2 Software Stack Analysis

In this section, we evaluate the resource requirements of the μPnP software stack in terms of memory footprint and runtime performance. Table 2 decomposes the μPnP software stack into individual elements and measures flash (ROM) and RAM requirements in absolute terms and as a percentage of the resources available on our evaluation platform, the ATMega128RFA1 [6].

	Flash (Bytes)	**RAM (Bytes)**
Peripheral Controller	2243 (1.7%)	465 (2.8%)
μPnP Virtual Machine	7028 (5.3%)	450 (2.7%)
ADC Native Library	2034 (1.5%)	268 (1.6%)
UART Native Library	466 (0.3%)	15 (0.1%)
I2C Native Library	436 (0.3%)	18 (0.1%)
μPnP Network Stack	2024 (1.5%)	302 (1.8%)
Total	14231 (10.8%)	1518 (9.2%)

Table 2. Detailed breakdown of μPnP's memory footprint

As can be seen from Table 2, μPnP consumes minimal memory both in absolute terms and relative to the available microcontroller memory resources.

Next, we measure the performance of the μPnP virtual machine in terms of executing driver bytecode instructions. We executed each bytecode instruction 500 times. On average, the execution of an instruction takes 39.7μs. This cost is further decomposed into the time required to execute *push()* and *pop()* stack operations. A push() operation takes on average 11.1μs, while a pop() operation requires 8.9μs. Finally, we measure the performance of the event router which

exchanges events between device drivers, native interconnect libraries and the network stack. The performance of the event router scales linearly in terms of number of events processed, and on average it takes 77.79 μs to process each event. Considered in sum, the VM execution and event dispatching performance evaluation demonstrates that the μPnP driver approach performs well even on embedded devices.

6.3 Driver DSL Analysis

In this section, we assess the effort required to develop μPnP drivers. We develop representative drivers for each of our four prototype peripherals and compare the Source Lines of Code (SLoC) and memory footprint of a μPnP DSL driver against a standard C driver. As can be seen from Table 3, μPnP drivers require fewer source lines of code and have a smaller memory footprint than standard C drivers. The large size discrepancy between different C device drivers is caused by the lack of hardware support for floating point operations on the ATMega128RFA1. As all floating point operations are executed in software, drivers involving floating point operations must include a software floating point library.

	μPnP DSL		**Native Variant**	
	(SLoC)	**(Bytes)**	**(SLoC)**	**(Bytes)**
TMP36 (ADC)	15	30	64	2956
HIH-4030 (ADC)	19	55	65	3304
ID-20LA RFID (UART)	43	150	89	592
BMP180 Pressure (I2C)	122	234	193	652
Average (examples)	50	117	103	1876

Table 3. Development efforts and memory footprint of device drivers

The key reason for the savings observed in Table 3 is that the μPnP DSL strongly separates the concerns of peripheral logic and platform-logic, resulting in fewer lines of code. Moreover, the compact bytecode representation of compiled μPnP drivers is shown in the reduced memory footprint. As μPnP bytecode is compact, it can be efficiently distributed across the network. On average μPnP drivers contain 52% fewer source lines of code and have a 94% smaller memory footprint.

6.4 Network Performance Analysis

In this section we analyse the performance of the μPnP networking stack for an uncongested one-hop network with low rates of packet loss. Table 4 provides performance timings for each network operation that occurs when a new peripheral is plugged in. All experiments were performed 10 times and averaged results are presented.

As can be seen from Table 4, peripheral discovery is fast, even for embedded platforms such as the ATMega128RFA1. Nevertheless, an analysis of multicast performance in multi-hop network topologies and unreliable network environments is left for future work.

	Average Time	**Standard Deviation**
Generate Multicast Address	2.59 ms	0.03 ms
Join Multicast Group	5.44 ms	0.01 ms
Request driver	53.91 ms	1.98 ms
Install 80 Byte Driver	59.50 ms	9.97 ms
Advertise Peripheral	45.37 ms	0.28 ms
Total time	188.53 ms	10.97 ms

Table 4. Detailed analysis of peripheral announcement and driver installation

7. Related Work

Plug-and-Play device integration for the IoT is composed of three phases. (i) device interconnection, (ii.) device driver development, and (iii.) service discovery and usage. We review the state-of-the-art in each of these areas in Section 7.1 and Section 7.3 respectively.

7.1 Peripheral Interconnection Approaches

Contemporary mainstream interconnection technologies like Universal Serial Bus (USB) [17] and IEEE 1394 (Fire Wire) [2] provide not only auto-configuration of peripheral devices, but also high-speed packet-based communication and device type identifiers. While high-speed communication with peripherals is typically unnecessary in IoT scenarios, device type identifiers are very useful, as they provide a mechanism for the embedded OS to discover peripherals and bootstrap the installation of necessary device drivers. Unfortunately, as shown in Section 6, even low power USB host chips consume too much power to be feasibly applied in IoT scenarios. The need for automation of peripheral configuration is particularly acute for the IoT, as a single IoT deployment may contain thousands of embedded devices.

Contemporary embedded interconnection technologies, focus on minimizing CPU, memory and energy consumption when connecting peripherals. Examples include: Universal Asynchronous Receiver/Transmitter (UART), Serial Peripheral Interface (SPI), Inter-Integrated Circuit (I2C) and System Management Bus (SMBus) [38]. UART and SPI are point-to-point connection technologies that connect two embedded devices, while I2C and the related SMBus assign each device an address and allow multiple peripherals to be daisy-chained on a low speed two-wire serial bus. The μPnP bus is capable of encapsulating all of these approaches, while providing support for device identification. This allows existing embedded peripherals using different interconnection technologies to be easily repackaged as μPnP devices.

Arduino [8] is a popular open-source embedded computing platform that introduces standard form-factor expansion boards, or *shields* that connect to the various communication pins on the Arduino mainboard and can therefore be used to add a wide range of peripherals, such as the USB

host shield that was used in our evaluation [28]. By making shield drivers available as libraries in the Arduino development environment, the complexity of integrating these peripherals is reduced. The popularity of the extensible Arduino platform illustrates the importance of supporting IoT peripherals. However, these drivers still need extensive manual configuration in order to operate correctly. To promote the adoption of μPnP we have realized our first prototype as an Arduino shield.

No embedded interconnection technology provides the device type identifiers that would allow the OS to autoconfigure the required driver software. μPnP provides such an approach using a simple design that is cost and energy efficient. This makes true Plug and Play peripheral integration available to IoT devices for the first time.

Mainstream approaches such as USB [17] or the Peripheral Component Interconnect (PCI) bus [34] encode device type and manufacturer data in peripherals using special purpose registers. This approach requires that peripherals embed a more complex circuit or chip to store identifier data. This raises the barrier to entry for third-party peripheral designers. In contrast, μPnP focuses on making peripheral identification as easy as possible for system integrators, and clearly connecting 4 resistors is much easier and cost-efficient than a dedicated circuit. On the other hand, we intend to explore how the vendor-ID + device-ID name structure of PCI and USB identification might serve as a good reference for μPnP. We discuss this in Section 9.

7.2 Embedded Driver Development

Platform-dependent driver code: The standard approach to implementing driver support in embedded Operating Systems such as TinyOS [41], Contiki [12] and SOS [16] is to expose common peripherals, such as the radio, ADC and UART via a well-known API that is provided by platform-specific libraries.

The Integrated Concurrency and Energy Management (ICEM) [23] device driver architecture is a core element of TinyOS 2.0 [41]. ICEM models device drivers as NesC components [13], which are extended with energy management code that allows TinyOS to more optimally schedule access to peripherals by applications, while preserving driver performance. As drivers are modelled as reusable software components, third-parties can safely download and reuse driver code at development time.

The Pixie OS [27] extends the NesC component model with support for resource-aware programming, wherein resource brokers mediate between low-level resources such as peripherals and application requirements, through the allocation of resource tickets. This provides fine-grained visibility of, and control over, resource allocation.

The prior approaches discussed above have two key shortcomings. First, drivers are written in a *low-level* and *platform-dependent* manner, which requires peripheral developers to be expert in multiple programming languages and operating systems. Second, they do not provide a clean separation of concerns between peripheral-specific logic and platform-specific logic, which precludes the reuse of driver code across heterogeneous platforms.

Platform-independence: Several project initiatives have investigated the notion of platform-independent drivers for mainstream systems. For example, the Uniform Driver Interface (UDI) project [7] aims to provide a well-defined and consistent driver interface across a range of hardware platforms and operating systems. In this way, UDI-enabled drivers work without modification on any supported system. This separation of driver and OS development yields various benefits in terms of development and maintenance effort. UDI shares a number of concepts with the μPnP driver language as it renders drivers independent from the underlying platform and operating system. However, while μPnP primarily focuses on embedded IoT devices, UDI was mainly built for mainstream systems, such as Linux or FreeBSD, in which resources are plentiful and peripheral hardware technologies are more diverse. As such, UDI focuses more on performance and supporting a rich variety of hardware technologies than efficiency.

Domain-specific languages: Several initiatives use domain specific languages to simplify driver development. The Devil [29] project focuses on simplifying device driver development by abstracting over low-level hardware complexities through a common Interface Definition Language (IDL). This IDL enables hardware communication, such as memory-mapped I/O or port-mapped I/O, to be described using high-level constructs instead of being written with low-level operations. The Devil compiler then checks safety properties and automatically generates all corresponding native code.

NDL [11] is similar to Devil in that it offers a domain-specific language for device drivers. This language provides high-level constructs for device programming, describing the driver in terms of its operational interface. Finally, HAIL (Hardware Access Interface Language) [39] adopts a domain-specific language that specifies attributes related to device access and generates device access functions for drivers on embedded systems. HAIL also generates run-time debugging code for driver developers.

Devil and NDL are designed for mainstream systems, however their concepts provide inspiration for developing the native interconnect libraries residing in the μPnP runtime environment. On the other hand, HAIL primarily targets embedded systems such as the IoT. It would therefore be interesting to explore how complementary concepts from HAIL such as run-time debugging might be applied in μPnP.

7.3 Remote Service Discovery

When integrating peripherals with distributed systems such as the IoT, it is necessary to advertise the services that the peripheral provides to the network and provide mechanisms to access those services. A number of service discovery pro-

tocols have been proposed to tackle this problem in mainstream computer systems including: Jini [5], UPnP [42] and SLP [14].

Jini [5] extends the Java Remote Method Invocation (RMI) interaction model to provide a modular and reusable mechanism for binding devices and services together. The Jini *registrar* supports the unicast or multicast discovery of software services by clients and returns a proxy object for the clients that allows for RMI access to the discovered service. Jini also supports proactive multicast announcement of service availability to remote clients. Jini discovery and announcement messages contain an extensible set of attributes that are defined by the software service. The registrar thus allows clients to discover and interact with services without prior knowledge of their location.

Universal Plug-and-Play (UPnP) [42] enables the platform-independent discovery of networked devices using the standard Web Services protocol stack. As with Jini, UPnP uses multicast to support the client-driven discovery and peripheral-driven advertisement of services. However, unlike Jini, UPNP is capable of operating in a decentralized fashion. A UPnP discovery request message contains a device type, unique identifier and a pointer to human-readable meta-data. A UPnP advertisement message provides a URL link from which peripheral meta-data can be retrieved. The UPnP description contains a list of interactions that can be performed with the discovered service. Discovery and advertisement messages use an event-based architecture which allows for transmission of HTTP messages over multicast UDP, while subsequent interactions between peripherals and clients use the standard Simple Object Access Protocol (SOAP). The related Devices Profile for Web Services (DPWS) protocol [40] defines a minimalist implementation of Web Services that supports resource-constrained devices.

Service Location Protocol (SLP) [14] is a simple UDP-based protocol for the discovery of networked peripherals. As with Jini and UPnP, SLP uses multicast communication and like UPnP, it operates in a decentralized fashion. All SLP peripherals periodically multicast an advertisement message, which is received and indexed by a Directory Agent (DA). Clients may then discover the indexed services by either multicasting a request to the DAs or, in the case of a network without DAs, by listening directly for multicast service advertisements from SLP peripherals. SLP discovery and advertisement messages contain a URL link to the related service and an extensible set of service attributes. Subsequent interaction between clients and discovered peripherals is outside of the scope of the SLP protocol, allowing for the use of diverse peripheral-specific protocols.

Prior approaches are *resource inefficient*: UPnP [42] and DPWS [40] use verbose XML data representations, while Jini [5] requires a complete Java Virtual Machine. Communication in SLP [14] is more efficient as it is based a custom UDP protocol. However, as SLP does not include device identifiers in multicast addresses additional filtering logic is required at the applications layer. The problem of *driver installation* is also not tackled by prior work. Instead, it is assumed that each device has all drivers pre-installed and that peripherals do not change over time. This is a particularly critical shortcoming for IoT deployments, which may contain many thousands of customizable nodes.

8. Conclusions

This paper introduced μPnP, a system for efficiently extending IoT devices with hardware peripherals in a plug and play fashion. μPnP achieves this vision through a systems approach involving: hardware, software and networking elements.

The μPnP hardware approach provides an energy-efficient way to identify embedded peripherals. Evaluation shows that this approach is far more energy efficient than USB. For example, in a realistic case, where peripherals are connected and disconnected once per hour, μPnP consumes over four orders of magnitude less energy than an embedded USB host controller.

The μPnP software approach provides a strong separation of concerns between peripheral logic, which is encapsulated in a platform-independent driver language, and platform-specific logic, which is provided by the μPnP runtime as native code. In comparison to standard C drivers, the μPnP DSL reduces source lines of code by an average of 52%, while the memory footprint of drivers is reduced by an average of 96%. Moreover, the μPnP software stack is small, consuming only 14231 bytes of flash memory and 1518 bytes of RAM for a typical IoT platform.

The μPnP network approach exploits the features of IPv6 multicast to realize an efficient service discovery protocol. Specifically, the complete peripheral discovery process, i.e. peripheral identification, driver installation and joining of multicast groups takes only 488.53 ms in a one-hop network.

Considered in sum, we believe that μPnP contributes a promising future vision for true Plug and Play customization of IoT devices.

9. Future Work

Our future work will proceed along four tracks: network experimentation, driver support, hardware improvements and design of the μPnP name space.

Networking: We are deploying a large network of μPnP devices across multiple geographic locations in order to test the performance of multicast service discovery in heterogeneous and multi-hop network environments. We also plan to investigate the use of *location-aware* multicast groups, to enable reasoning over both classes of device and their physical locations.

Driver Support: In terms of the μPnP driver repository, we plan to significantly expand the range of sensors supported by μPnP, while making the resulting driver code pub-

licly available in a single online location. We will also investigate automated approaches to validating third-party driver software. This will ensure that the μPnP address space remains scalable.

Hardware Design: We are currently designing a new revision of the μPnP controller board, which is cheaper, smaller and therefore suitable for integration with a wider range of devices. This revision will remain backwards compatible with the design presented in this paper.

μPnP Name Space: We are investigating how the μPnP name-space should be re-designed. Our approach will, on the one hand be inspired by the ID structure of PCI and USB, which includes a vendor ID and device ID. However we hope to go further, for example by embedding hierarchical device typing and security information.

Acknowledgments

This research is partially funded by the Research Fund KU Leuven and the IOF-TRANSITION project.

References

[1] IEEE Standard for a Simple 32-Bit Backplane Bus: NuBus. *ANSI/IEEE Std 1196-1987*, 1988.

[2] IEEE Standard for a High-Performance Serial Bus. *IEEE Std 1394-2008*, pages 1–954, Oct 2008.

[3] J. Abley and K. Lindqvist. Operation of Anycast Services. RFC 4786 (Best Current Practice), Dec. 2006.

[4] Analog Devices. Low Voltage Temperature Sensors Datasheet. `http://www.analog.com/static/imported-files/data_sheets/TMP35_36_37.pdf`. [Accessed: Oct 2014].

[5] K. Arnold. *The Jini specification*. Jini technology series. Addison Wesley, 1999.

[6] Atmel. AVR ATMegaRFA128 Datasheet. `http://www.atmel.com/Images/doc8266.pdf`. [Accessed: Oct 2014].

[7] R. Barned and R. Richards. Uniform driver interface (udi) reference implementation and determinism. In *Real-Time and Embedded Technology and Applications Symposium, 2002. Proceedings. Eighth IEEE*, pages 301–310, 2002.

[8] S. Barrett. *Arduino Microcontroller Processing for Everyone!: Third Edition*. Synthesis Lectures on Digital Circuits and Systems. Morgan & Claypool Publishers, 2013.

[9] Bosch. BMP180 Digital barometric pressure sensor Datasheet. `http://ae-bst.resource.bosch.com/media/downloads/pressure/bmp180/Flyer_BMP180_08_2013_web.pdf`. [Accessed: Oct 2014].

[10] N. Brouwers, K. Langendoen, and P. Corke. Darjeeling, a feature-rich vm for the resource poor. In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, SenSys '09, pages 169–182, New York, NY, USA, 2009. ACM.

[11] C. L. Conway and S. A. Edwards. Ndl: A domain-specific language for device drivers. In *Proceedings of the 2004 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES '04, pages 30–36, New York, NY, USA, 2004. ACM.

[12] A. Dunkels, B. Gronvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks*, LCN '04, pages 455–462. IEEE Computer Society, 2004.

[13] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesc language: A holistic approach to networked embedded systems. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, pages 1–11, 2003.

[14] E. Guttman. Service location protocol: Automatic discovery of ip network services. *IEEE Internet Computing*, 3(4):71–80, July 1999.

[15] B. Haberman and D. Thaler. Unicast-Prefix-based IPv6 Multicast Addresses. RFC 3306 (Proposed Standard), Aug. 2002.

[16] C.-C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava. A dynamic operating system for sensor nodes. In *Proceedings of the 3rd International Conference on Mobile Systems, Applications, and Services*, MobiSys '05, pages 163–176, 2005.

[17] Hewlett-Packard, Intel, Microsoft, Renesas, ST-Ericsson, Texas Instruments. Universal serial bus specification revision 3.1. `http://www.usb.org/developers/docs/`. [Accessed: Oct 2014].

[18] Honeywell. HIH-4030/31 Series Humidity Sensors Datasheet. `http://sensing.honeywell.com/honeywell-sensing-hih4030-4031%20series-product-sheet-009021-4-en.pdf`. [Accessed: Oct 2014].

[19] ID-innovations. ID-3LA, ID-12LA, ID-20LA Low Voltage Series Reader Moudules Datasheet. `http://id-innovations.com/httpdocs/ID-3LA,ID-12LA,ID-20LA.pdf`. [Accessed: Oct 2014].

[20] Intel Corporation. Isa bus specification and application notes. `http://ia601609.us.archive.org/15/items/bitsavers_intelbusSpep89_3342148/Intel_ISA_Spec2.01_Sep89.pdf`. [Accessed: Oct 2014].

[21] International Electrotechnical Commission (IEC). *Preferred number series for resistors and capacitors*. Multiple. Distributed through American National Standards Institute (ANSI), 1963.

[22] S. Kawamura and M. Kawashima. A Recommendation for IPv6 Address Text Representation. RFC 5952 (Proposed Standard), Aug. 2010.

[23] K. Klues, V. Handziski, C. Lu, A. Wolisz, D. Culler, D. Gay, and P. Levis. Integrating concurrency control and energy management in device drivers. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 251–264, 2007.

[24] J. Koshy, I. Wirjawan, R. Pandey, and Y. Ramin. Balancing computation and communication costs: The case for hybrid execution in sensor networks. *Ad Hoc Networks*, 6(8):1185 – 1200, 2008. Energy Efficient Design in Wireless Ad Hoc and Sensor Networks.

[25] P. Levis and D. Culler. Maté: A tiny virtual machine for sensor networks. *SIGOPS Oper. Syst. Rev.*, 36(5):85–95, Oct. 2002.

[26] Logos Electromechanical. Zigduino r2 Manual. http://wiki.logos-electro.com/zigduino-r2-manual. [Accessed: Oct 2014].

[27] K. Lorincz, B.-r. Chen, J. Waterman, G. Werner-Allen, and M. Welsh. Resource aware programming in the pixie os. In *Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems*, SenSys '08, pages 211–224, 2008.

[28] Maxim Integrated Datasheet. http://datasheets.maximintegrated.com/en/ds/MAX3421E.pdf. [Accessed: Oct 2014].

[29] F. Mérillon, L. Réveillère, C. Consel, R. Marlet, and G. Muller. Devil: An idl for hardware programming. In *Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation - Volume 4*, OSDI'00, pages 2–2, Berkeley, CA, USA, 2000. USENIX Association.

[30] Motorola, Inc. SPI Block Guide. http://www.ee.nmt.edu/~teare/ee308l/datasheets/S12SPIV3.pdf. [Accessed: Oct 2014].

[31] NXP Semiconductors. I2C-bus specification and user manual. http://www.nxp.com/documents/user_manual/UM10204.pdf. [Accessed: Oct 2014].

[32] G. Oikonomou and I. Phillips. Stateless multicast forwarding with RPL in 6LowPAN sensor networks. In *IEEE International Conference on Pervasive Computing and Communications Workshops (PERCOM Workshops 2012)*, pages 272–277, March 2012.

[33] A. Osborne. *An Introduction to Microcomputers: Basic concepts*. An Introduction to Microcomputers. Osborne/McGraw-Hill, 1980.

[34] PCI-SIG. PCI Local Bus Specification Revision 3.0. https://www.pcisig.com/specifications/conventional/pci_30/. [Accessed: Mar 2015].

[35] Seeed Studio. Introduction to Grove. http://www.seeedstudio.com/document/pdf/Introduction%20to%20Grove.pdf. [Accessed: Oct 2014].

[36] Z. Shelby and C. Bormann. *6LoWPAN: The Wireless Embedded Internet*. Wiley Series on Communications Networking & Distributed Systems. Wiley, 2011.

[37] Y. Shi, K. Casey, M. A. Ertl, and D. Gregg. Virtual machine showdown: Stack versus registers. *ACM Trans. Archit. Code Optim.*, 4(4):2:1–2:36, Jan. 2008.

[38] SMBus Specification Working Group. System Management Bus (SMBus) Specification. http://smbus.org/specs/. [Accessed: Oct 2014].

[39] J. Sun, W. Yuan, M. Kallahalla, and N. Islam. Hail: A language for easy and correct device access. In *Proceedings of the 5th ACM International Conference on Embedded Software*, EMSOFT '05, pages 1–9, New York, NY, USA, 2005. ACM.

[40] The OASIS Group. OASIS Devices Profile for Web Services (DPWS). http://docs.oasis-open.org/ws-dd/ns/dpws/2009/01, July 2009. [Accessed: Oct 2014].

[41] The TinyOS 2.x Working Group. Tinyos 2.0. In *Proceedings of the 3rd International Conference on Embedded Networked Sensor Systems*, SenSys '05, pages 320–320, 2005.

[42] UPnP Forum. UPnP Device Architecture. http://upnp.org/specs/arch/UPnP-arch-DeviceArchitecture-v1.1.pdf. [Accessed: Oct 2014].

[43] T. Winter, P. Thubert, A. Brandt, J. Hui, R. Kelsey, P. Levis, K. Pister, R. Struik, J. Vasseur, and R. Alexander. RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks. RFC 6550 (Proposed Standard), Mar. 2012.